The Data-Oriented Programming attack

Table of Contents

The attack described in paper *Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks* exploits the ProFTPd vulnerability CVE-2006-5815 to read the private SSL-key. Authors of the paper have created a Metasploit exploit module and a virtual machine image containing the vulnerable version of the software and instructions on how to run and exploit it. These are the main sources and basis of this analysis in addition to the original paper.

The vulnerability and the exploit is described in the next section. The idea of the attack is that logic of the vulnerable function is so expressive that local variables can be used to direct the execution flow enough to execute arbitrary memory operations. Attacker has control over the local variables, due to the stack buffer overflow vulnerability. Attacker is build by triggering simple memory operations carefully in the correct order, one at a time to achieve the desired outcome. The goal is to read the secret SSL key and this requires following pointers in a chain starting from a global static variable `ssl_ctx` and exfiltrating the data.

ProFTPd vulnerability CVE-2006-5815

ProFTPd 1.3.0 has a feature that allows customized messages to be shown when user enters a directory. The messages are saved in .message file in each directory and they can be created and edited by any user having write-access to the directory. Message files can contain special character sequences which are replaced with dynamic content like time and date, username, server name or name of the current directory. The dynamic content is filled in function `pr_display_file` which calls `sreplace` to replace the patterns with actual values.

The vulnerability is introduced by an off-by-one comparison allowing a buffer to overflow with attacker controlled data. The buffer is allocated from the stack giving the attacker full control of all stack variables as shown in *Table 1*.

Table 1. Stack variables in function `sreplace`:

type	name	address	description
`(char **)`	`src`	`0xbfbedd8c`	source string, contents of .message
`(char **)`	`cp`	`0xbfbedd88`	current pointer to the pbuf
`(char ***)`	`mptr`	`0xbfbedd84`	item of marr
`(char ***)`	`rptr`	`0xbfbedd80`	item of rarr
`(char **)`	`pbuf`	`0xbfbedd7c`	pointer to the buffer (buf)
`(size_t *)`	`mlen`	`0xbfbedd78`	length of string mptr
`(size_t *)`	`blen`	`0xbfbedd74`	size of buf
`(int *)`	`dyn`	`0xbfbedd70`	(irrelevant)
`(size_t *)`	`rlen`	`0xbfbedd64`	length of string rptr
`(char **)`	`m`	`0xbfbedd60`	(irrelevant)
`(char **)`	`r`	`0xbfbedd5c`	(irrelevant)
`(va_list *)`	`args`	`0xbfbedd58`	(irrelevant)
`(char *(*)[33])`	`marr`	`0xbfbedcd4`	match string array
`(char *(*)[33])`	`rarr`	`0xbfbedc50`	replacement string array
`(char (*)[4096])`	`buf`	`0xbfbecc50`	destination buffer

Listing 2. Call to `sreplace` in `pr_display_file`

```
outs = sreplace(p, buf,
  "%C", (session.cwd[0] ? session.cwd : "(none)"),
  "%E", main_server->ServerAdmin,
  "%F", mg_size,
  "%f", mg_size_units,
```

```
 6  "%i", total_files_in,
 7  "%K", mg_xfer_bytes,
 8  "%k", mg_xfer_units,
 9  "%L", serverfqdn,
10  "%M", mg_max,
11  "%N", mg_cur,
12  "%o", total_files_out,
13  "%R", (session.c && session.c->remote_name ?
14    session.c->remote_name : "(unknown)"),
15  "%T", mg_time,
16  "%t", total_files_xfer,
17  "%U", user,
18  "%u", session.ident_user,
19  "%V", main_server->ServerName,
20  "%x", session.class ? session.class->cls_name : "(unknown)",
21  "%y", mg_cur_class,
22  "%z", mg_class_limit,
23  NULL);
```

Listing 2 shows the `sreplace` call in `pr_display_file` with all replacements listed as function arguments. In function `sreplace` (partly presented in *Listing 4*) first arrays `marr` and `rarr` are constructed. All patterns present in the `src` string are added to the `marr` array and the replacements are added to the `rarr` array. Length of the final string is also calculated in the process. If certain conditions are met, the stack buffer will be used instead of allocating space from the heap. The exact logic behind this decision appears to be unexpected: buffer is allocated from the heap if the final size of the string is less than `BUFSIZ`, if the size is `BUFSIZ` or more, `BUFSIZ` is used as the maximum length and a stack buffer will be used. The stack buffer has size of 4096 bytes in the demonstration binaries. In rest of this report, it is assumed that the stack buffer is used.

Listing 4. Snippet from `sreplace`

```
 1if (!pbuf) {
 2  cp = pbuf = buf;
 3  dyn = FALSE;
 4  blen = sizeof(buf);
 5}
 6
 7while (*src) {
 8  for (mptr = marr, rptr = rarr; *mptr; mptr++, rptr++) {
 9    mlen = strlen(*mptr);
10    rlen = strlen(*rptr);
11
12    if (strncmp(src, *mptr, mlen) == 0) {
13      sstrncpy(cp, *rptr, blen - strlen(pbuf));
14      if (((cp + rlen) - pbuf + 1) > blen) {
15        pr_log_pri(PR_LOG_ERR,
16                   "WARNING: attempt to overflow internal ProFTPD buffers");
17        cp = pbuf + blen - 1;
18        goto done;
19
20      } else {
21        cp += rlen;
22      }
23
24      src += mlen;
25      break;
26    }
27  }
28
29  if (!*mptr) {
30    if ((cp - pbuf + 1) > blen) {
31      pr_log_pri(PR_LOG_ERR,
32                 "WARNING: attempt to overflow internal ProFTPD buffers");
33      cp = pbuf + blen - 1;
34    }
35    *cp++ = *src++;
36  }
37}
```

The replacing is done in a loop that iterates through all characters in `src` string. On every iteration, the substring is compared to each replacement pattern in `marr` with `strncmp`. When a match is found, `sstrncpy` is used to copy the replacement from `rptr` to the destination buffer: `sstrncpy(cp, *rptr, blen - strlen(pbuf));`. The last argument of `sstrncpy` is the maximum length of the copied integer which is expected to be an unsigned integer. Value of this argument is calculated from size of the buffer (4096 bytes) and current length of its contents which should prevent buffer overflows.

When `src` points to a character that does not start a matching substring, the character itself is copied. A check is done before the copy to detect attempts to write outside of the buffer bounds. The checking code is shown in *Listing 4* in lines 30-35. When writing to the last item of the buffer, `cp` points to `pbuf[4095]`, so `(cp - pbuf + 1) > blen` will become 4095 + 1 > 4096 and the write is allowed. This is also illustrated in *Listing 3*. In C strings are terminated by a zero byte. All standard string functions, like `strlen`, rely on this. When the last character in the buffer is overwritten with a non-zero byte, the string it not properly terminated inside the buffer.

Listing 3. Buffer overflow check

```
1(cp - pbuf + 1) > blen
2
3// cp-pbuf is the length of data in the buffer
4// +1 is added for the NULL terminator
5// blen is 4096
```

```
//
//    ,------------------- blen ------------------,
//
//    |AAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB_|
//    ^                                           ^
//    |                                           |
//    |                                          cp
//  pbuf
```

On the next iteration, `strlen(pbuf)` will be bigger than the actual buffer due to the missing null-terminator, causing the result of `blen - strlen(pbuf)` to be negative. Last parameter of `sstrncpy` is unsigned `size_t n` and the negative value is interpreted as a large integer. Hence this invocation of `sstrncpy` overflows outside buffer bounds into the stack and overwrites local variables.

Desired stack contents depend on the attack. Overwriting the return address in the stack would compromise the control flow when the function returns. However, the data-oriented programming attack only manipulates the local variables.

The vulnerability is exploited as follows:

1. Two directories are created: `dir1` and its subdirectory `dir2`. Length of name of `dir1` is 90 bytes and `dir2` 233 bytes. Total length is hence 323 bytes. The directory names contain the data that will overwrite stack contents. Directory names are not allowed to contain characters `*`, `/`, LF (`\n`), 0xff or 0x00, because these have special meaning.
2. Contents of the `.message` file are: `%C` repeated 12 times, character `C` 300 times and `%C` again 40 times. Following command is used in the Metasploit module to generate file contents

   ```
   filedata = "\x25C" * 12 + 'C' * 300 + "\x25C" * 40
   ```

3. The file is uploaded to the FTP server under `dir2`.
4. Processing of `.message` is triggered by changing the working directory to `dir2` with `CWD` command. Function `sreplace` is called from `pr_display_file` with contents of `.message` file.
5. Every `%C` is replaced with the name of the current working directory. After the 12 first replacements the buffer is around 4000 bytes full. Next 300 characters are copied without replacement, but with the character-by-character copying. The NULL byte string terminator gets removed here. The first replacement of the rest `%C` sequences will trigger the overflow and stack contents are replaced with the string `/dir1/dir2` where `dir1` and `dir2` are the names of the directories.

Attack description

Relevant variables and memory locations

The attack relies on some known memory addresses and abuses them to keep state and have some temporary storage. Most of them depend only depend on the compiled binary. They remain same on every execution and attacker can extract them from the binary.

The proof-of-concept attack uses and requires knowledge of the following addresses. See *Listing 7* for details about the memory mappings.

Variables

- `ssl_ctx` (`0x80eb0a8`), pointer to the SSL context, a global static variable, which can not be accessed from outside the source file, declared in `mod_tls.c`
- `mons` (`0x080dc6c0`), string pointer array, a local static variable declared in `support.c`
- `session`, a global variable declared in `main.c`
  - `session.total_bytes_out` (`0x80edbfc`)
- `main_server` (`0x080e3dec`), global pointer to a `server_rec` structure, declared in `dirtree.c`
- `resp_buf` (`0x80e6000`), a global static buffer declared in `response.c`

Other memory locations

- `NON_EMPTY_HIGH` (e.g. `0xb74e9001`), any address of non-null byte in stack or library area
- `NON_EMPTY_LOW` (`0x0804b3be`), a non-empty string in code region
- `G_PTR_ADDR` (`0x080e0410`), address of a pointer to global writable data, located in the code section
- `G_PTR` (`0x080e0430`), address of an unused area in the data section
- `EMPTY` (`0x0804b3b9`), address of an empty string (a zero byte)
- `DATA_START` (`0x080dc4b1`), address of some empty area in the data section
- `BYTE_MAP` (`0x080bxxx`), a map of addresses in the code section where each byte (from 0x01 to 0xff) can be found

Listing 7. Memory mappings of the process

```
08048000-080db000 r-xp 00000000 08:01 3024990    /home/dop/dop-attacks/proftpd-1.3.0/proftpd
080db000-080dc000 r--p 00092000 08:01 3024990    /home/dop/dop-attacks/proftpd-1.3.0/proftpd
080dc000-080e2000 rw-p 00093000 08:01 3024990    /home/dop/dop-attacks/proftpd-1.3.0/proftpd
080e2000-080ee000 rw-p 00000000 00:00 0
084f4000-08534000 rw-p 00000000 00:00 0          [heap]
08534000-08535000 rw-p 00000000 00:00 0          [heap]
08535000-08536000 rw-p 00000000 00:00 0          [heap]
```

```
b7338000-b7343000 r-xp 00000000 08:01 143633     /lib/i386-linux-gnu/libnss_files-2.15.so
b7343000-b7344000 r--p 0000a000 08:01 143633     /lib/i386-linux-gnu/libnss_files-2.15.so
b7344000-b7345000 rw-p 0000b000 08:01 143633     /lib/i386-linux-gnu/libnss_files-2.15.so
b7345000-b7346000 rw-p 00000000 00:00 0
b7346000-b735a000 r-xp 00000000 08:01 132091     /lib/i386-linux-gnu/libz.so.1.2.3.4
b735a000-b735b000 r--p 00013000 08:01 132091     /lib/i386-linux-gnu/libz.so.1.2.3.4
b735b000-b735c000 rw-p 00014000 08:01 132091     /lib/i386-linux-gnu/libz.so.1.2.3.4
b735c000-b735d000 rw-p 00000000 00:00 0
b735d000-b7360000 r-xp 00000000 08:01 143615     /lib/i386-linux-gnu/libdl-2.15.so
b7360000-b7361000 r--p 00002000 08:01 143615     /lib/i386-linux-gnu/libdl-2.15.so
b7361000-b7362000 rw-p 00003000 08:01 143615     /lib/i386-linux-gnu/libdl-2.15.so
b7362000-b7505000 r-xp 00000000 08:01 143623     /lib/i386-linux-gnu/libc-2.15.so
b7505000-b7507000 r--p 001a3000 08:01 143623     /lib/i386-linux-gnu/libc-2.15.so
b7507000-b7508000 rw-p 001a5000 08:01 143623     /lib/i386-linux-gnu/libc-2.15.so
b7508000-b750b000 rw-p 00000000 00:00 0
b750b000-b7517000 r-xp 00000000 08:01 132042     /lib/i386-linux-gnu/libpam.so.0.83.0
b7517000-b7518000 r--p 0000b000 08:01 132042     /lib/i386-linux-gnu/libpam.so.0.83.0
b7518000-b7519000 rw-p 0000c000 08:01 132042     /lib/i386-linux-gnu/libpam.so.0.83.0
b7519000-b76ac000 r-xp 00000000 08:01 131986     /lib/i386-linux-gnu/libcrypto.so.1.0.0
b76ac000-b76bb000 r--p 00193000 08:01 131986     /lib/i386-linux-gnu/libcrypto.so.1.0.0
b76bb000-b76c2000 rw-p 001a2000 08:01 131986     /lib/i386-linux-gnu/libcrypto.so.1.0.0
b76c2000-b76c5000 rw-p 00000000 00:00 0
b76c5000-b7717000 r-xp 00000000 08:01 132152     /lib/i386-linux-gnu/libssl.so.1.0.0
b7717000-b7719000 r--p 00051000 08:01 132152     /lib/i386-linux-gnu/libssl.so.1.0.0
b7719000-b771d000 rw-p 00053000 08:01 132152     /lib/i386-linux-gnu/libssl.so.1.0.0
b771d000-b771e000 rw-p 00000000 00:00 0
b771e000-b7726000 r-xp 00000000 08:01 143630     /lib/i386-linux-gnu/libcrypt-2.15.so
b7726000-b7727000 r--p 00007000 08:01 143630     /lib/i386-linux-gnu/libcrypt-2.15.so
b7727000-b7728000 rw-p 00008000 08:01 143630     /lib/i386-linux-gnu/libcrypt-2.15.so
b7728000-b774f000 rw-p 00000000 00:00 0
b775f000-b7761000 rw-p 00000000 00:00 0
b7761000-b7778000 r-xp 00000000 08:01 143628     /lib/i386-linux-gnu/libpthread-2.15.so
b7778000-b7779000 r--p 00016000 08:01 143628     /lib/i386-linux-gnu/libpthread-2.15.so
b7779000-b777a000 rw-p 00017000 08:01 143628     /lib/i386-linux-gnu/libpthread-2.15.so
b777a000-b777e000 rw-p 00000000 00:00 0
b777e000-b777f000 r-xp 00000000 00:00 0          [vdso]
b777f000-b779f000 r-xp 00000000 08:01 143631     /lib/i386-linux-gnu/ld-2.15.so
b779f000-b77a0000 r--p 0001f000 08:01 143631     /lib/i386-linux-gnu/ld-2.15.so
b77a0000-b77a1000 rw-p 00020000 08:01 143631     /lib/i386-linux-gnu/ld-2.15.so
bf9a5000-bf9c6000 rw-p 00000000 00:00 0          [stack]
```

Operations

Execution of the attack consists of small operations. The primitive operations are `copy`, `read`, `deref` and `add`. First one copies data from address to another, second reads data from an address and returns it, third dereferences a pointer and the last performs integer addition. Some more powerful operations are built on top of these four. The write operation uses the copy and `BYTE_MAP` to write arbitrary data.

copy

The copy-operation relies on known global addresses as they are the only known ones. Arbitrary source and destination addresses are given to it as parameters and they are written to the stack in the desired places. Most other variables are just set so that arithmetics, references and comparisons work in a desirable way and do not result in memory violations like segmentation faults. An additional requirement is that stack must not contain any of the forbidden characters like `/` or the null byte.

By exploiting the stack buffer overflow vulnerability in `sreplace` function, the content of the stack is arranged as follows.

- `rarr[0]` is a valid, but irrelevant address, it is referenced by `strlen`
- `rarr[1]` is the source address of the copy
- rest of `rarr` contents are irrelevant for the attack
- `marr[0]` = `NON_EMPTY_HIGH`
- `marr[1]` = `EMPTY`
- rest of `marr` contents are irrelevant for the attack
- `blen` = destination - `NON_EMPTY_HIGH` + 1
- `mlen` + `src` = `NON_EMPTY_HIGH`
- `pbuf` = `NON_EMPTY_HIGH`
- `mptr` = `NON_EMPTY_HIGH`
- `cp` of the next iteration is the destination address, `rlen` is added to `cp`, so destination address is `cp+rlen`
- other variables (`rptr`, `args`, `r`, `m`, `dyn`) do not affect the execution

Listing 5. Execution of the copy operation

```
1//                                                    iteration 1       iteration 2.0          iteration 2.1
2
3  while (*src) {                                     //                src=non_empty+1
4    for (mptr = marr, rptr = rarr; *mptr; mptr++, rptr++) {          //                marr[0]=non_empty      marr[1]=emptystr
5                                                     //                rarr[0]=validptr       rarr[1]=copy_src
6      mlen = strlen(*mptr);                          //                mlen>0                 0
7      rlen = strlen(*rptr);                          //                rlen=0                 strlen(copy_src)
8                                                     //
9      if (strncmp(src, *mptr, mlen) == 0) {          //                no match               mlen=0, take branch
10       sstrncpy(cp, *rptr, blen - strlen(pbuf));    // stack overflow                        copies the data
11       if (((cp + rlen) - pbuf + 1) > blen) {       // not taken
12         pr_log_pri(PR_LOG_ERR,                     //
13           "WARNING: attempt to overflow internal ProFTPD buffers");//
14         cp = pbuf + blen - 1;                      //
15         goto done;                                 //
16                                                    //
```

```
17        } else {                                                    //
18          cp += rlen;                                               // cp+rlen=copy_dest
19        }                                                           //
20                                                                    //
21        src += mlen;                                                // src+mlen=non_empty+1
22        break;                                                      // break from for loop
23      }                                                             //
24    }                                                               //
25                                                                    //
26    if (!*mptr) {                                                   // mptr=non_empty, not taken
27      if ((cp - pbuf + 1) > blen) {                                 //
28        pr_log_pri(PR_LOG_ERR,                                      //
29          "WARNING: attempt to overflow internal ProFTPD buffers"); //
30        cp = pbuf + blen - 1;                                       //
31      }                                                             //
32      *cp++ = *src++;                                               //
33    }                                                               //
34  }                                                                 //
```

read

Read operation is executed in multiple steps. As a preparation, contents of an array called `mons` are overwritten. Originally `mons` contains 12 pointers to strings of month names. It is a local array in the `pr_strtime` function. The array is filled with an address `G_PTR` which is a pointer to global writable memory. This is done only once and all subsequent reads rely on this.

Read itself works mostly like copy, but the destination address is an address of a writable global `G_PTR`. Stack contents are almost identical to the copy operation. `marr[1]` is empty and "source address" `rarr[1]` is the address being read. When executed, the exploit writes data to `G_PTR`.

After executing the exploit, a `.message` file with `%T` is created. Reading the file triggers `pr_strtime`, which creates a response with date string that contains also the month string from the `mons` array. All pointers in this array are replaced with `G_PTR` so instead of the month name, the previously copied data is returned.

Listing 6. Beginning of `pr_strtime`, declaration of `mons`

```
1const char *pr_strtime(time_t t) {
2  static char buf[30];
3  static char *mons[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
4    "Aug", "Sep", "Oct", "Nov", "Dec" };
5  static char *days[] = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

deref

The dereferencing operation dereferences the value currently located in the `ServerName` field of the `main_server` strucure. Replacement string `%V` is used for this. `main_server.ServerName` is modified with the copy operation before each dereference, so the dereference operation itself does not need to exploit the vulnerability. It only creates a `.message` file with a string containing `%V`. Then `main_server->ServerName` will be copied to a known position in response buffer `resp_buf`. After triggering the dereference, the data can be copied from `resp_buf` to the final destination.

write_string

`write_string` operation is used to write arbitrary string to arbitrary memory location. It uses the copy operation to write one byte at a time from a known location in the process memory.

add

The FTP session has `session.total_bytes_out` counter which is used to keep track of total amount of downloaded bytes. This integer can be incremented without exploiting any vulnerabilities, because the attacker can upload and download files to the server. To add number x to the counter, attacker uploads a file of x bytes and then downloads it. Ability to increment arbitrary values can be achieved by using the copy operation before addition to move the value that will be incremented to `session.total_bytes_out`. After the addition the value can be copied back.

Attack flow

The whole attack which reads and returns the private RSA key can be broken up to following steps.

1. Overwrite 12 pointers in the `mons`-array with pointers to a global, writable location (G_PTR). As a result all functions that print month names will print the value pointed by `G_PTR`.
    - calls the copy operation 12 times with `mons` as destination and `G_PTR_ADDR` as source
2. Write known address of `ssl_ctx` to `DATA_START` (empty area in data section)
    - write_string
        - copy x 4
3. Copy `aaa\0` to `DATA_START[4:7]`
    - write_string x 3

        - copy x 3
    - write0

4. Read `main_server` global
    - read
5. Copy 6 bytes from `DATA_START` to `main_server.ServerName`
    - copy
6. Dereference the `ssl_ctx`-pointer
    - deref
7. Copy dereferenced value to `add_loc` (`SESSION_TOTAL_BYTES_OUT`)
    - copy: single zero-byte to `d+4`
    - copy: actual data from `d` to `add_loc`
8. Add offset of the field to address of `SSL_CTX`
    - add 0xb0
9. Copy the result from `session.total_bytes_out` to `main_server.ServerName`
    - copy: data from `add_loc` to `ptr`
10. Repeat steps 6-9 to follow the reference chain
    - step 8 may be omitted if field offset is 0
11. Read the address
    - read
12. Write zeros to `main_server.tcp_recvbuf_len`
    - unclear why
13. Read the data from the address extracted in step 11
    - repeat `read(d_val+i*4,4)` 64 times

Memory accesses and scope violations

Stack buffer overflow

The unexpected memory accesses that enable exploitation do not break any variable scoping rules. The first access for for the exploit is overwriting the last character of the buffer which is completely acceptable normally. The 4096th byte (`buf[4095]`) is overwritten in support.c:730 (line 35 in *Listing 4*) after the previous call to `sstrncpy` has filled the buffer.

The second access is writing outside the bounds of the buffer in the `sstrncpy` function. Destinations of these writes are also in the scope of the calling function, because they are other stack variables. `buf[4096]` is written when calling `sstrncpy` with large `n` value due to the integer overflow explained earlier.

DOP attack

To be able to read data from arbitrary pointers, the attack manipulates the `mons` array. `mons` is declared inside `pr_strtime` function in support.c making it inaccessible from outside that function in C, but because it is a static variable it will be placed in the global data section.

All twelve string pointers in the `mons` array are replaced with `G_PTR`, a global pointer. Address of `G_PTR` is located in `G_PTR_ADDR` which resides in the code section. This is referenced on each of the twelve copies inside the `sstrncpy`.

copy

Copy operation changes all local variables to manipulate data pointers and control the execution flow.

1. After the overflow `mptr` is dereferenced with `!*mptr`. At this point `mptr` points to the stack or library section to a non-zero byte. The attack requires this address to start with b. This is not even a real variable an not inside the scope of the function.

    ```
    if (!*mptr) {
    ```

2. Next violation happens on the next iteration of the while loop. The condition dereferences local pointer `src`. To make sure the execution continues, `src` points to non-empty plus 1.

    ```
    while (*src) {
    ```

3. The string where `*mptr` now points is referenced in `strlen`. The same is done for `*rptr`. `*mptr` points here to the non-empty string. `*rptr` points to nowhere important, it is just a valid pointer `0x0804802f`. Both references violate the scopes.

   ```
   mlen = strlen(*mptr);
   rlen = strlen(*rptr);
   ```

4. The parameters to `strncmp` are `src`, `*mptr`. The comparison will read at least first characters of the strings. `src` points to the second character of the non-empty string and `*mptr` points to the same non-empty string, but to its beginning.

   ```
   if (strncmp(src, *mptr, mlen) == 0) {
   ```

   Second iteration of the for loop is taken next and now `mptr = marr[1]` and `rptr = rarr[1]`.

5. `*mptr` points to empty string which is located in process code section. `*rptr` is the arbitrary source pointer. `strlen` is called on both.

6. The `strncmp` tests compares then `*mptr` to `src`. `src` still points to some part of the non-empty string.

7. The data is then copied from the arbitrary pointer `*rptr` to `cp` with `sstrncpy`. Whether this operation violates any variable scopes depends on the actual pointers. The pointers that are used are listed in the beginning of the attack description. The variables are also listed in *Attack flow* section.

read

Read works almost like copy, but with destination address `G_PTR`, a constant global address. Read also includes practically same violations as the copy. The read value is returned in the response by using `%T` pattern which is replaced with string representation of the current time built in `%T`. The violation happens in `pr_strtime` where static array `mons` is dereferenced. Normally the pointers in this array point to strings "Jan", "Feb" etc., but all these pointers have been replaced with `G_PTR_ADDR`. This pointer is passed to `snprintf` which reads data from it.

deref

Dereferencing uses the `%V` replacement pattern. This triggers a dereference of `main_server->ServerName` in `pr_display_file` function in display.c:216. This does not violate any rules. The pointer has however been replaced earlier with copy.

Summary and conclusions

The vulnerability is a stack buffer overflow. The DOP attack overwrites all data in the stack which allows it to gain full control of the execution. The attack also relies on global memory references and known addresses to a large extent. Most important dereferences are not done in the vulnerable function, but in the callees like `sstrncpy`.

To avoid violating scopes when referencing the values, some of the known addresses could be replaced by addresses of variables that are actually inside the scope of the function like globals that are passed to `sreplace` as function arguments. Feasibility of this approach would require more analysis.

The attack would be stopped by enforcing variable scopes runtime. The attack references many memory addresses that do not even contain any variables. These references would probably be easy to detect and stop. However, deducing the variables belonging to the scope may be complicated. Especially argument passing and mutable global data structures make the scope very dynamic. The call chain is quite long and the vulnerable function takes variable arguments. Data structures like linked lists may make the scope quite large and harder to maintain.